GIT

Graphs

A. Ada, K. Sutner
Carnegie Mellon University

Spring 2018

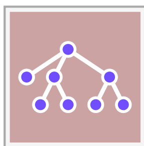
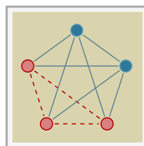
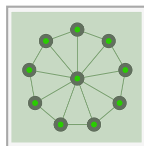
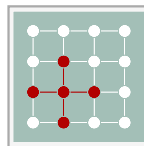
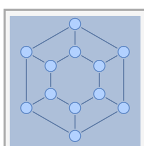
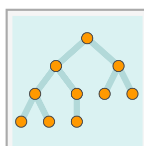
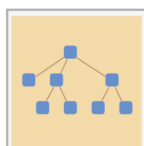
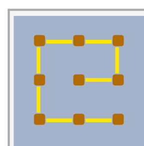
Outline

1 Graphs

2 Representation

3 Path Existence

4 BFS and DFS

Ancient History

A quote from a famous mathematician (homotopy theory):

> Combinatorics (read: graph theory) is the slums of topology.
>
> J. H. C. Whitehead

In the early 20th century "combinatorics" was a label for everything discrete and really outside of classical mathematics.

Not a good sign.

Less Ancient History

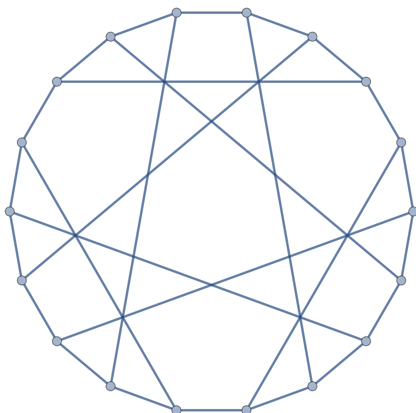
Things have improved slightly since.

> We have not begun to understand the relationship between combinatorics and conceptual mathematics.
>
> J. Dieudonné (1982)

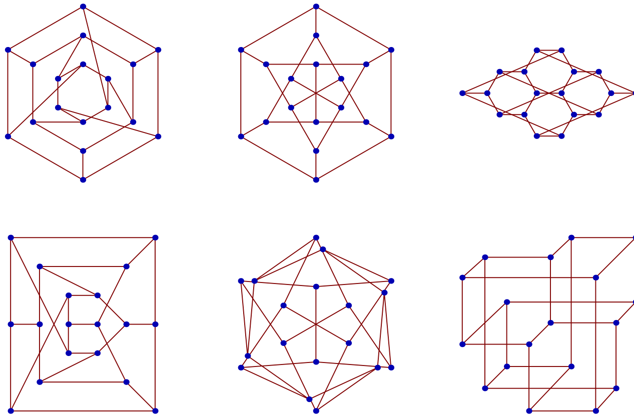
A nicely underhanded compliment.

Still, a lot of combinatorics (and graph theory) is highly algorithmic, so it naturally fits in the framework of computation: in contrast, say, classical differential equations are quite problematic. For example, there is currently no compelling theory of computation on the reals.

Pappus



But Beware:



Degrees

Counting the number of neighbors of a vertex turns out to be important, so there is some special terminology.

Definition

Let $G = \langle V, E \rangle$ be a digraph and $u \in V$ a vertex.

The out-degree of u is

$$\text{odeg}(u) = |\{ z \in V \mid (u, z) \in E \}|$$

The in-degree of u is

$$\text{ideg}(u) = |\{ z \in V \mid (z, u) \in E \}|$$

The degree of u is the sum of out-degree and in-degree.

In a ugraph the degree of a vertex is defined by

$$\deg(u) = |\{ z \in V \mid \{u, z\} \in E \}|$$

Basic Counting

Proposition

A digraph on n vertices has at most n^2 edges.

A ugraph on n vertices has at most $n(n+1)/2$ edges if one allows self-loops.

A ugraph without self-loops has at most $n(n-1)/2$ edges.

So the number of edges is $O(n^2)$.

One often has to distinguish between sparse graphs where the number of edges is much smaller than n^2 (say, something like $O(n \log n)$) and dense graphs where it is close to n^2.

For graph algorithms, dependency of running time on the number of edges is usually the critical question.

Proposition

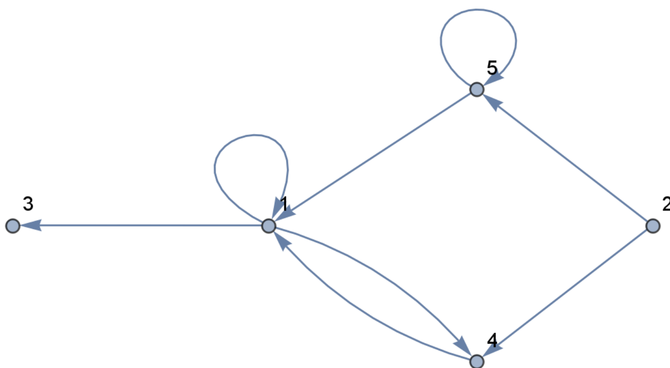
In any ugraph, $\sum \deg(u) = 2|E|$.

As a consequence, the number of odd-degree vertices must be even.

Proposition

In any digraph, $\sum \text{odeg}(u) = \sum \text{ideg}(u) = |E|$.

- Graphs
- 2 Representation
- Path Existence
- BFS and DFS



$n = 5$ vertices and $m = 8$ edges (2 self-loops).

Representation: Data Structures

edge list list of pairs of integers

$$(1, 1), (4, 1), (1, 4), (2, 4), (5, 1), (1, 3), (2, 5), (5, 5)$$

adjacency list array of linked lists

```
1:  1, 3, 4
2:  4, 5
3:  -
4:  1
5:  1, 5
```

adjacency matrix square Boolean matrix

$$\begin{matrix} 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{matrix}$$

The Standard Representations

Suppose $\langle V, E \rangle$ is a digraph on n vertices and m edges. It is often convenient to assume that $V = [n]$.

Definition

The edge list representation of a digraph consists of a list of length m of ordered pairs.

The sorted edge list representation of a digraph consists of a sorted list of length m of ordered pairs.

The adjacency list representation of a digraph consists of an array A of length n of lists: $A[u]$ is a list of all $v \in V$ such that $(u, v) \in E$.

The adjacency matrix representation of a digraph consists of a Boolean matrix B of size $n \times n$: $B[u, v] = 1 \iff (u, v) \in E$.

Exercises

Exercise

Concoct conversion algorithms that translate between any two of these representations. What are the time complexities?

Exercise

Define the digraph $G^{\mathrm{op}} = \langle V, E^{\mathrm{op}} \rangle$ by flipping all edges in G:
$(u, v) \in E^{\mathrm{op}} \iff (v, u) \in E$.

Explain how to compute G^{op} in all representations. What is the time complexity of your algorithms?

Memory Requirement

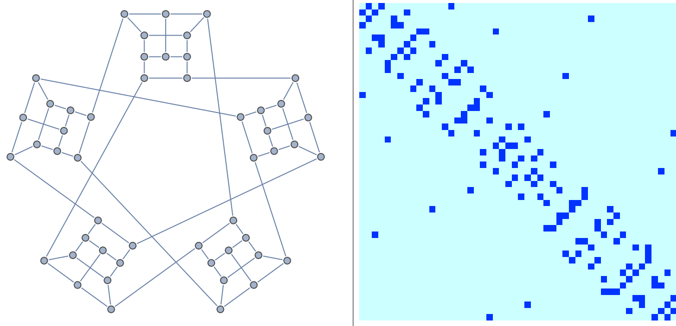
Suppose we have n vertices and m edges, so that $m \leq n^2$.

Sizes of the data structures:

- (sorted) edge list: $\Theta(m)$
- adjacency list: $\Theta(n + m)$
- adjacency matrix: $\Theta(n^2)$ (but smaller constants)

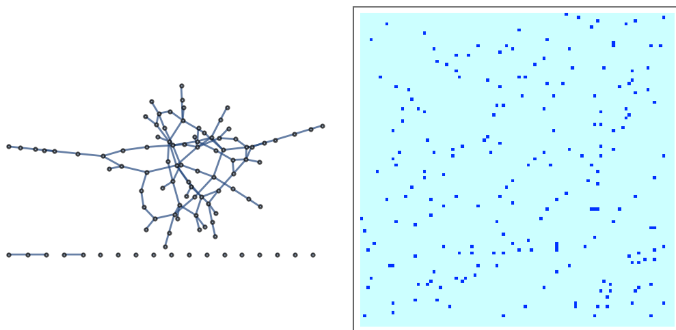
For sparse graphs, plain adjacency matrices are problematic: the size of the data structure does not adjust to the actual size of the graph.

Watkins Snark



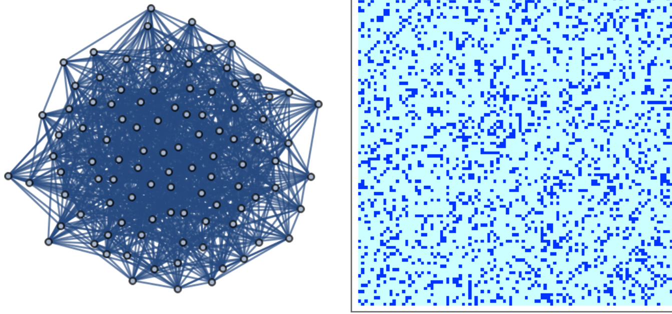
snark $n = 50$ and $m = 75$

Sparse



random $n = m = 100$

Not So Sparse



random $n = 100$ and $m = 1000$

But Beware ...

Any realistic implementation of adjacency matrices uses packed arrays: an integer is used to represent, say, 64 bits (bit-parallelism).

This approach produces a slight overhead for a query "is (u, v) and edge" but it allows one to obtain adjacency information about 64 vertices in "one step".

Asymptotically adjacency matrices are no match for adjacency lists, but for some reasonable values of n (think a few hundred, depending very much on hardware) they can be faster.

If you know for some reason that your algorithm will never touch graphs beyond, say, size 512 you are probably better off with a lovingly hand-coded adjacency matrix implementation (in a real language like C).

Sparse Matrices

As an aside: matrix implementations can be competitive even if the vertex set is large provided that

- the graph is sparse, and
- the implementation is based on sparse matrices.

A sparse matrix implementation does not require $\Theta(n^2)$ storage but $\Theta(m)$: only the non-zero entries in the matrix are associated with storage. This approach requires fairly messy pointer-based data structures and is quite difficult to implement.

Some modern computational environments (like Mathematica) use sparse matrices as the default implementation of a graph.

Local Queries

Given two vertices u and v, the most elementary question we can ask is:

Is $(u, v) \in E$?

The cost of answering this query is

- edge list: $O(m)$,
- sorted edge list: $O(\log m)$,
- adjacency list: $O(\text{odeg}(u))$,
- adjacency matrix: $O(1)$.

But note that this is just time, we are ignoring space.

Visit thy Neighbor

There are many graph algorithms that require a visit to all the neighbors of a particular vertex. So we have a code fragment of the form

```
1       vertex u, v;
2
3       u = ....;
4
5       foreach (u,v) edge do
6               ... v ...
```

For example, this is how one computes out-degrees.

Neighbor Traversal

This type of traversal takes

- edge list: $\Theta(m)$,
- sorted edge list: $O(\log m + \text{odeg}(u))$,
- adjacency list: $O(\text{odeg}(u))$,
- adjacency matrix: $\Theta(n)$.

In other words, an adjacency list implementation works very well, but neither edge lists nor adjacency matrices are generally suitable for this algorithmic task.

Example: Cycle Testing

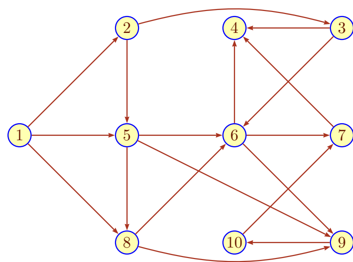
Suppose we want to check whether a digraph is acyclic.

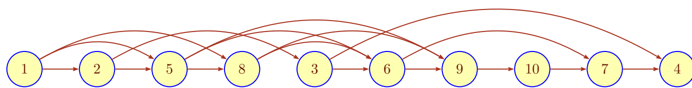
We will solve a slightly harder problem known as topological sorting:

- Given a digraph G, return NO if G has a cycle.
- Otherwise return a permutation $u_1, u_2, \ldots, u_n$ of the vertex set such that $(u_i, u_j) \in E$ implies $i < j$.

In other words, we want to arrange the vertices along a line so that all edges go from left to right. Note that the algorithm returns a "certificate of acyclicity".

Example



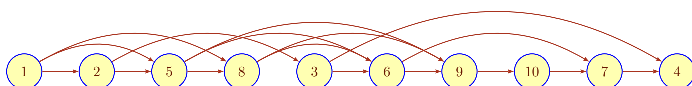


Ranks

A good way of representing the permutation is to compute a rank for each vertex in the graph: rank 1 means the vertex is leftmost, rank 2 is the second from the left and so on.

In the last example, the ranks are

x	1	2	3	4	5	6	7	8	8	9	10
rk(x)	1	2	5	10	3	6	9	4	8	7	8



Warm-Up

Proposition

Let G be a digraph such that every vertex in G has in-degree at least 1. Then G contains a cycle.

Proof.

Show by induction on k that G contains a path of the form

$$x_k, x_{k-1}, x_{k-2}, \ldots, x_1, x_0$$

where x_0 is chosen arbitrarily.

The induction step uses the fact that x_k has in-degree at least 1.

For $k = n$ we must have a repeated vertex on this path, and hence a cycle in G.

□

Sanity Check

Proposition

A digraph admits a topological sort if, and only if, it is acyclic.

Proof.

It is clear that a graph with a topological sort must be acyclic.

For the opposite direction, argue by induction on the number of vertices.

Since G is acyclic it must have an in-degree 0 vertex u.

Set $\text{rk}(u) = 1$ and continue with $G - u$.

□

Proof to Algorithm

The proof yields a recursive "algorithm" with outline

```
1   topsort( digraph G ) {
2
3       if( n==1 )  done;
4
5       find u in V s.t.  indeg(u)==0;
6       rank[u] = rk++;
7       H = G - u;
8       topsort( H );
9   }
```

Why the quotation marks?

Towards a Real Algorithm

The idea that we compute a subgraph

$$H = G - u$$

in line 7 amounts to algorithmic suicide: building a new data structure for H will require some $O(n + m)$ steps. The search in line 5 is also uncomfortable.

It is much better to simply mark vertex u as being removed.

Of course, we need to make sure that the in-degrees are changed accordingly and keep track of candidates for removal.

Real TopSort Algorithm


```
1     Stack     Z;
2
3     forall v in V
4             if( indeg[v]==0 ) Z.push(v);
5
6     while( !Z.empty() ) {
7          x = Z.pop();
8          rank[x] = rk++;
9          forall (x,y) in E {          // neighbor traversal
10                indeg[y]--;
11                if( indeg[y]==0 ) Z.push(y);
12         }
13    }
```

Exercises

Exercise

Implement topological sorting.

Exercise

How would you compute the in-degree of all vertices in a digraph, in all representations?

Exercise

How would you check if a ugraph has a triangle: edges $\{a, b\}, \{b, c\}, \{c, a\}$ where a, b and c are distinct vertices?

- Graphs
- Representation
- 3 Path Existence
- BFS and DFS

Paths and Reachability

In a graph G, node t is reachable from node s if there is a path from s to t in G.

Problem: **Reachability**
Instance: A labeled graph G, nodes s and t
Question: Is there a path from s to t in G?

Problem: **Bounded Reachability**
Instance: A labeled graph G, nodes s and t, a bound B.
Question: Is there a path from s to t in G of length at most B?

Note that if t is reachable then there is a path of length at most $n - 1$.

Notation

We'll often write

$$\pi : u \to v$$

to indicate that π is a path from u to v and $|\pi|$ for its length.

The distance from u to v is

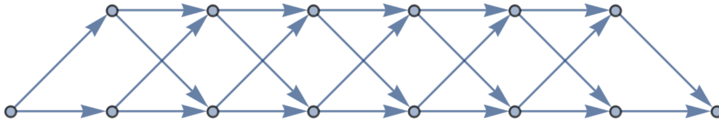
$$\text{dist}(u, v) = \min\big(|\pi| \mid \pi : u \to v \big)$$

If there is no path, this is assumed to be ∞.

Duh

Since we are only dealing with finite graphs, Reachability is clearly decidable.

But brute-force algorithms based on enumeration of all paths are going to be exponential, even when the graph is acyclic.



Boolean Matrix Multiplication

How can we solve Bounded Reachability for $B = 2$?

If we use Boolean matrices to represent E there is a really simple answer. Recall that the product $C = A \cdot B$ of two Boolean matrices is defined by

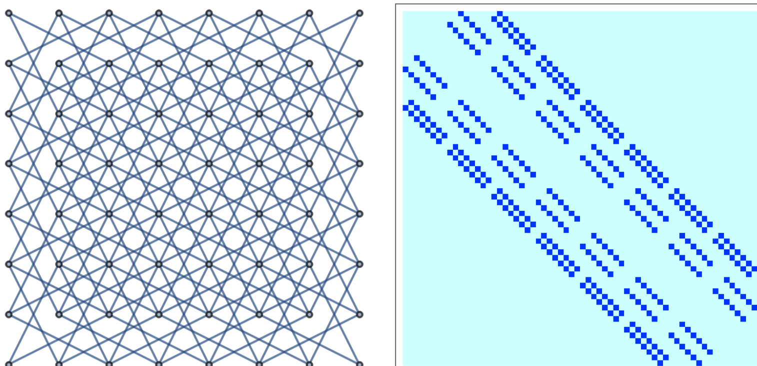
$$C(i,j) = \bigvee_k A(i,k) \wedge B(k,j)$$

We have emphasized the logical operations here, but often this would simply be written as $C(i,j) = \sum_k A(i,k) \cdot B(k,j)$ (standard Boolean algebra).

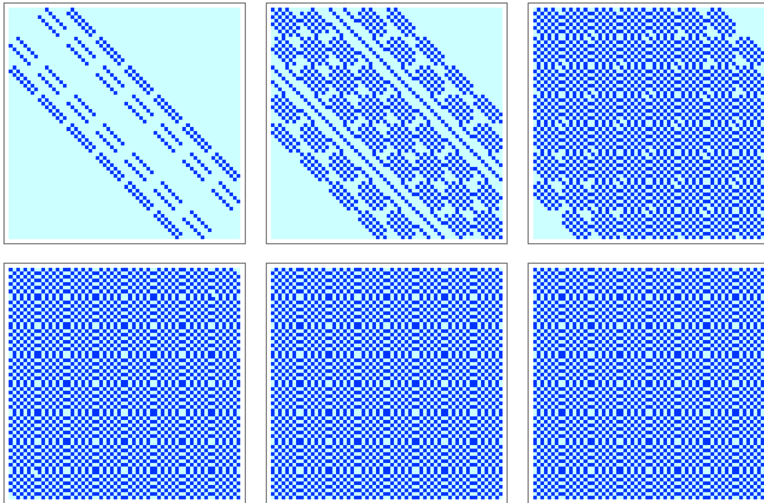
Proposition

Let A be the adjacency matrix of G. Then $A^k(u,v) = 1$ iff there is a path $\pi : u \to v$ of length k in G.

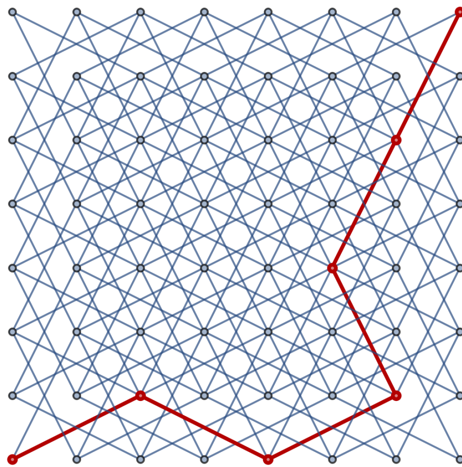
Knights



All Knights



All Knights



Back To Reachability

To solve Reachability, we essentially need to compute the set of all reachable points:

$$R(s) = \{ v \in V \mid \exists \pi : s \to v \}$$

For the time being, we will deal with reachability for all source vertices s.

Define the reachability matrix $A^\star$, a Boolean matrix, by:

$$A^\star(u, v) = 1 \iff \exists \pi : u \to v$$

Clearly, $A^\star$ is the sum over all the matrices

$$I, A, A^2, A^3, \ldots, A^k, \ldots$$

Exercises

Exercise

Why can the sequence $I, A, A^2, A^3, \ldots, A^k, \ldots$ of Boolean matrices for knight moves not end in a fixed point? What is the period?

Exercise

What would happen with rook, bishop, ... instead of a knight?

Exercise

Could there be a piece with period 3?

Reachability and BMM

We are dealing with finite graphs, so the sequence

$$I, A, A^2, A^3, \ldots, A^k, \ldots$$

must be ultimately periodic. But then the whole sum is actually

$$A^\star = I + A + A^2 + A^3 + \ldots + A^\ell$$

for some finite ℓ. In fact, by simple geometry we have

$$A^\star = I + A + A^2 + A^3 + \ldots + A^{n-1}$$

Alas, computing $A^\star$ this way takes time $O(n)$ BMMs.

Speedup

Let $B = A + I$ (reflexive closure). Then

$$B^k = I + A + A^2 + A^3 + \ldots + A^k$$

So we only need to compute B^{n-1}.

Using fast exponentiation (repeated squaring) we can do this in $\log n$ BMMs.

- Graphs
- Representation
- Path Existence

4. BFS and DFS

Inductive Structure

A totally different way to compute the set of reachable points $R(s)$ is to think of it as being inductively defined. We are now interested in a single source vertex.

- s is in $R(s)$.
- If u is in $R(s)$ and $(u, v) \in E$ then v is in $R(s)$.

This suggests to start with a "approximation" $R = \{s\}$ and keep adding the targets of edges whose source is already in R.

Prototype Algorithm

We build an approximation R to $R(s)$ in stages.

Let's say $(u, v) \in E$ requires attention (at some particular point during the execution of the algorithm) if $u \in R$ but $v \notin R$.

```
1
2         R = { s };
3
4         while( some edge (u,v) requires attention )
5                 add v to R;
6
7         return R;
8
```

Correctness 1

Proposition

At any time during the execution, $R \subseteq R(s)$.

Proof.

It is easy to check that $R \subseteq R(s)$ is a loop invariant:

Initially $R = \{s\} \subseteq R(s)$.

Assume $R \subseteq R(s)$ and (u, v) requires attention.

Then there is a path from s to u.

But then there also is a path from s to v.

□

Correctness 2

Proposition

Upon completion, $R(s) \subseteq R$.

Proof.

Proof is by induction on the distance k form s to $v \in R(s)$.

$k = 0$: $s \in R$ by initialization.

$k > 0$: Consider shortest path $s = x_0, x_1, \ldots, x_{k-1}, x_k = v$.

By IH, x_{k-1} is placed into R at some point. But then the edge (x_{k-1}, v) requires attention unless v is already in R. In either case, v winds up in R.

□

Bookkeeping

We have to organize the order in which edges (u, v) are handled: usually several edges will require attention, we have to select a specific one.

To this end it is best to place new nodes into a special container C (rather than just in R): C holds all the vertices that might be the source of an edge requiring attention.

```
1
2       R = C = { s };
3
4       while( C not empty )
5           u = pick in C and remove;
6           if( some edge (u,v) requires attention )
7               add v to R, C;
8
9       return R;
10
```

Implementation

R should be a bit-vector (or perhaps a hash table) so we can check in $O(1)$ time if an edge requires attention.

C must support constant time inserts and select-remove operations (but not search). A stack or queue will work fine.

We can traverse the adjacency list to find all edges that might require attention.

Theorem

The running time of the prototype algorithm is $O(n + m)$.

Exercise

Explain the running time of the prototype algorithm more carefully.

Exploration

Using a stack for S we obtain a concrete exploration algorithm.

```
1
2     explore_graph( vertex s )
3     {
4          S.push( s );
5          put s into R;               // R a set
6
7          while( S not empty ) {
8                  x = S.pop();
9                  forall  (x,y) in E  do
10                    if( y not in R ) {
11                        S.push(y);
12                        put y into R;
13                    }
14         }
15    }
```

Depth First Search (DFS)

One advantage of using a stack is that we can avoid implementing it explicitly: recursion will take care of it.

```
1     dfs( vertex x )
2     {
3             put x into R;
4             forall (x,y) in E  do
5                    if( y not in R )
6                        dfs( y );
7     }
```

Note that the additional space requirement is $n + O(n)$: n for the bit-vector R and $O(n)$ for the recursion stack.

Breadth First Search (BFS)

Alternatively, we can use a queue to implement the container.

```
1      bfs( vertex s )
2      {
3          Q.enqueue( s );
4          put s into R;                // R a set
5
6          while( Q not empty ) {
7                x = Q.dequeue();
8                forall  (x,y) in E  do
9                   if( y not in R ) {
10                      Q.enqueue(y);
11                      put y into R;
12                  }
13         }
14     }
```

Reachable Set

Let G be a digraph on n vertices and m edges.

Assume that we have an adjacency list representation of the graph.

Theorem

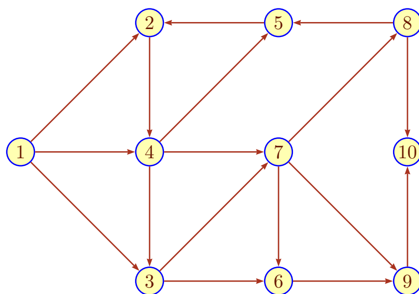
One can compute the set of all vertices reachable from a given vertex in time $O(n+m)$.

Corollary

One can check whether there is a path from a given source to a given target vertex in time $O(n+m)$.

The running times are correct as stated, but note that the details of a run of DFS or BFS depend very much on the order of vertices in the adjacency lists. This little feature can make correctness proofs rather messy.

Example



A digraph on 10 vertices and 17 edges.

Adjacency Lists

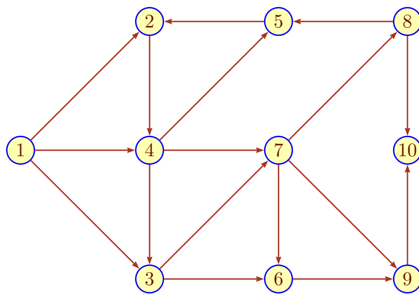
To describe a run of DFS or BFS we need to fix a concrete adjacency list representation.

```
1  1: 4 2 3       6: 9
2  2: 4           7: 6 9 8
3  3: 7 6         8: 10 5
4  4: 3 5 7       9: 10
5  5: 2          10: -
```

In this graph $R(1) = [10]$ but the order of discovery differs in DFS and BFS.

```
1  vertex:  1  2  3  4  5  6  7  8  9 10
2
3  DFS:     1 10  3  2  9  5  4  8  6  7
4
5  BFS:     1  3  4  2  5  7  6  9  8 10
```

Example



$R(1) = [10]$.

$R(6) = \{6, 9, 10\}$, $R(9) = \{9, 10\}$ and $R(10) = \{10\}$.

For all other x: $R(x) = \{2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

Spanning Trees

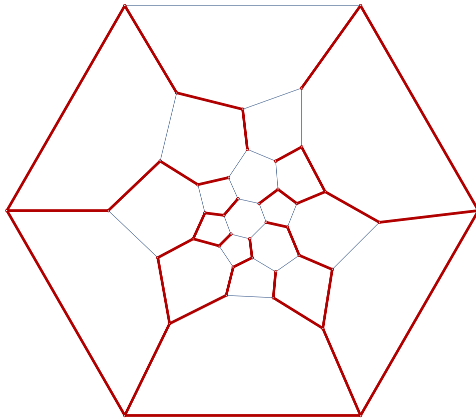
Suppose $G = \langle V, E \rangle$ is a connected ugraph. A spanning tree of G is a subgraph $T = \langle V, E' \rangle$ that is connected and acyclic.

In other words, we remove edges from G until only a tree is left.

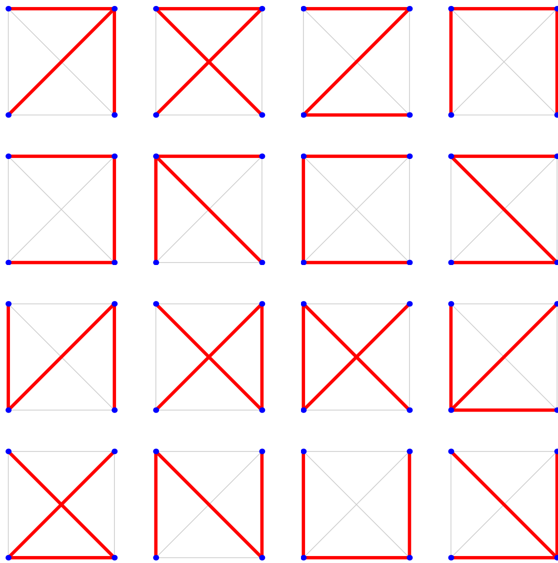
This is particularly interesting if we have a cost function $\pi : E \to \mathbb{R}_+$ and we want a spanning tree that minimizes total cost, as so-called minmimum spanning tree.

More later.

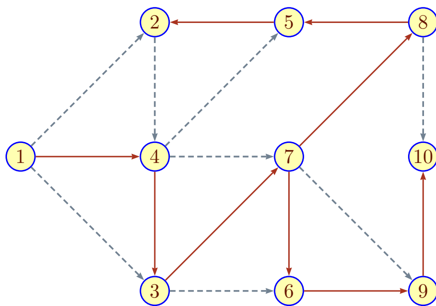
Fullerene



K_4 Spanning Trees

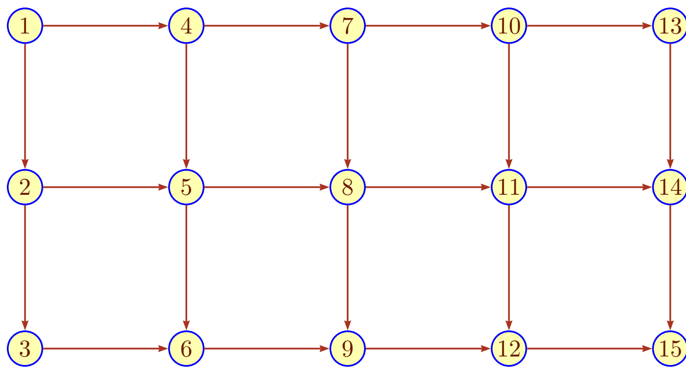


The DFS Tree



The edges receiving attention in DFS form a spanning tree (if the graph is connected; otherwise we get a tree for each connected component). This DFS-generated spanning tree is the starting point for many other graph algorithms.

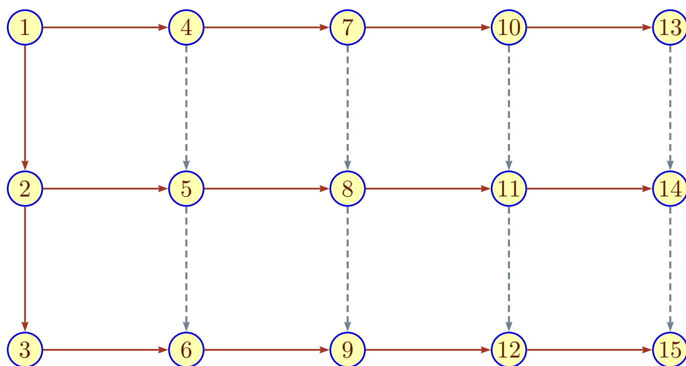
Grid Example



Suppose adjacency lists are ordered.

What DFS tree do we get if we start the search at vertex 1?

The Tree



Typical DFS behavior: run off to a distant vertex, then backtrack – but run off again at the first opportunity.

Exercises

Exercise

What tree would BFS produce on this grid graph?

Exercise

Characterize all the digraphs for which DFS and BFS produce the same spanning tree.

(I once knew the answer, but I forgot.)